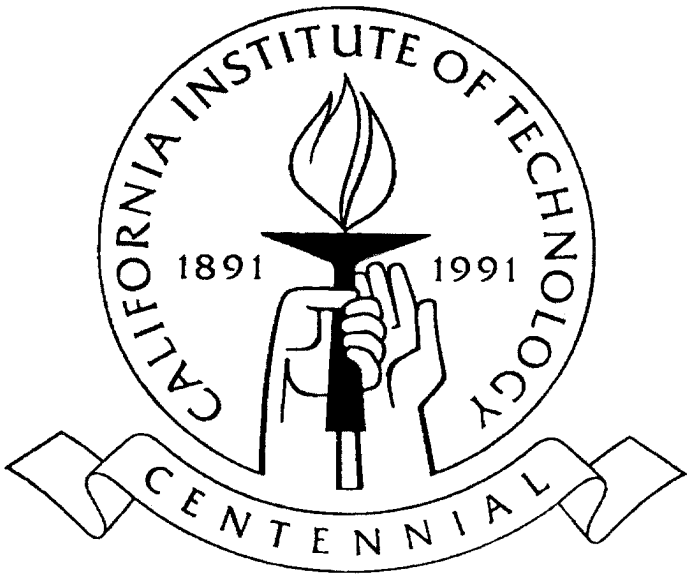

A Distributed Implementation of a Task Pool

H. Peter Hofstee
Johan J. Lukkien
Jan L.A. van de Snepscheut

Computer Science Department
California Institute of Technology

Caltech-CS-TR-91-05

A Distributed Implementation of a Task Pool

H. Peter Hofstee, Johan J. Lukkien, Jan L. A. van de Snepscheut
Computer Science
California Institute of Technology
Pasadena, CA 91125

21 March 1991

In this paper we present a distributed algorithm to implement a task pool. The algorithm can be used to implement a processor farm, i.e., a collection of processes that consume tasks from the task pool and possibly produce tasks into it. There are no restrictions on which process consumes which task nor on the order in which tasks are processed. The algorithm takes care of the distribution of the tasks over the processes and ensures load balancing. We derive the algorithm by transforming a sequential algorithm into a distributed one. The transformation is guided by the distribution of the data over processes. First we discuss the case of two processes, and then the general case of one or more processes.

Keywords: load balancing, processor farm, distributed computation.

1. Program notation

For the sequential part of the algorithms, we use Edsger W. Dijkstra's guarded command language [1]. For statements $S0$ and $S1$, statement $S0||S1$ denotes their concurrent execution. The constituents $S0$ and $S1$ are then called processes. The statements may share variables (cf. [5]). We transform our algorithms in such a way, however, that the final code contains no shared variables and all synchronization and communication is performed by message passing. The semantics of the message passing primitives is as described in [4]. The main difference with C.A.R. Hoare's proposal in [2] is in the naming of channels rather than processes. In [3], the same author proposes to name channels instead of processes in communication commands, but differs from our notation by using one name per channel instead of our two: output command $R!E$ in one process is paired with input command $L?v$ in another process by declaring the pair (R, L) to be a channel between the two processes. Each channel is between two processes only. When declaring (R, L) to be a channel, we write the name on which the output actions are performed first and the name on which the input actions are performed last.

For an arbitrary command A, let $\mathbf{c}\,A$ denote the number of completed A actions, i.e., the number of times that command A has been executed since initiation of the program's execution. The *synchronization requirement* (cf. [4]) fulfilled by a channel (R, L) is that

$$\mathbf{c}\,R = \mathbf{c}\,L$$

holds at any point in the computation.

The execution of a command results either in the completion of the action or in its suspension when its completion would violate the synchronization requirement. From suspension until completion an action is *pending* and the process executing the action is delayed. We introduce boolean $\mathbf{q}\,A$ equal to the predicate "an A action is pending". Associated with a channel (R, L) we use $\overline{R} \equiv \mathbf{q}R!$ and $\overline{L} \equiv \mathbf{q}L?$ as defined in [7]. $\overline{L}$, pronounced *probe of* L, evaluates to *true* if an $R!$ action is pending, and to *false* otherwise. The *progress requirement* states that actions are suspended only if their completion would violate the synchronization requirement, i.e., channel (R, L) satisfies

$$\neg\mathbf{q}\,R \ \vee \ \neg\mathbf{q}\,L \qquad .$$

The nth R action is said to match the nth L action. The completion of a matching pair of actions is called a *communication*. The *communication requirement* states that execution of matching actions $R!E$ and $L?v$ amounts to the assignment $v := E$. The semantics of the **if** statement is reinterpreted (cf. [2]). If all guards in an **if** statement evaluate to *false*, its execution results in suspension of the execution rather than **abort**. When the guards contain no shared variables, an if statement that is suspended remains suspended and therefore this definition is compatible with the original semantics.

2. Statement of the problem

Given is a finite, constant number of processes $\mathbf{C}_{(0\leq i<N)}$ capable of consuming or producing tasks. Associated with each of these processes is a pool process $\mathbf{P}_i$. Each process $\mathbf{C}_i$ communicates only with its pool process. The pool processes also communicate with one another. Each pool can hold an unbounded number of tasks. (We will address bounded pool sizes in section 6.) The problem is to construct pool processes in such a way that no process $\mathbf{C}_i$ is idle forever if there is enough work to be done.

We do not want to impose restrictions on the processes $\mathbf{C}_i$. Since a process $\mathbf{C}_i$ might, once activated, produce another task at any time, we cannot expect the algorithm for the pool to terminate. The requirements for the program can be formulated as follows:

(0). A situation in which a process is idle whereas there are enough tasks in the network to keep all processes busy does not persist.

(1). If the processes $\mathbf{C}_i$ do not consume or produce tasks, the number of communications is bounded.

The first requirement guarantees progress, whereas the second requirement guarantees quiescence in the absence of communications between the $\mathbf{C}_i$ and $\mathbf{P}_i$.

Coming up with the proper formalization of the first condition is nontrivial. On the one hand the condition should not be overly restrictive, since this will give rise to a large communication overhead. On the other hand several candidates for this condition had to be rejected because they allowed stable states that were undesirable. We finally replaced the first condition with the following:

(0)'. Either communications are pending or $(\forall i :: p_i \leq T) \ \vee \ (\forall i :: p_i \geq T)$.

(0)". There is no deadlock.

T is some fixed positive number, the threshold, and p_i denotes the number of elements in pool i. If we assume items are communicated one at a time, the solution seems obvious; select a pair of pools where one has more than T and the other has less than T elements and send an element from the fuller to the emptier pool. Written as a global sequential program, and ignoring the processes $\mathbf{C}_i$ and the actual communication of tasks, we obtain the program in Figure 1. The quantified expression ($[\!]i :: guarded\text{-}command_i$) should be interpreted as one guarded command for every i.

$$\textbf{do } ([\!]\ i,j :: p_i < T \ \wedge\ p_j > T \rightarrow p_i, p_j := p_i + 1, p_j - 1\ )\ \textbf{od}$$

–Figure 1–

The negation of the guards is $(\forall i :: p_i \leq T) \ \vee\ (\forall i :: p_i \geq T)$. Termination is guaranteed by the bound function $\sum_i |p_i - T|$. This sequential algorithm can be transformed into a distributed algorithm on a fully connected graph. However, since the resulting algorithm would not scale well with the size of the graph, we do not pursue this solution here.

We have tried to solve this problem by an approach similar to that in [6], that is, by examining a linear communication network for the processes $\mathbf{P}_i$ and by introducing variables that will approach $(\exists j : 0 \leq j < i : p_j < T)$ etc.. This gives rise to an extremely complicated algorithm if we do not want to make assumptions about the $\mathbf{C}_i$, mainly because any global information is volatile.

3. An asymmetric solution

If the process graph is not fully connected, the problem cannot be solved in a symmetric way on the basis of information about a process and its neighbors only. To see this, consider the following series of values for the pools:

$$p_0 = T - 1 \qquad p_1 = T \qquad p_2 = T + 1$$

A symmetric bound function such as $\sum_i |p_i - T|$ would not be decreased by communicating an item from pool $\mathbf{P}_2$ to pool $\mathbf{P}_1$ or from pool $\mathbf{P}_1$ to pool $\mathbf{P}_0$. Stated differently, a solution that would do this cannot be expected to satisfy our second condition: we cannot put an upperbound on the number of communications.

The solution is as follows. We break the symmetry by choosing one node, and constructing a rooted, directed, acyclic graph (RDAG) with this one node as its root. Nodes are assigned a nonnegative weight in such a way that there is at least one path from each node to the root node where the weights of the nodes on the path form a strictly decreasing sequence. One possible way of doing this is by a depth-first search from the root. We define $p\text{-}edge_{ij} \equiv w_i > w_j \ \wedge\ edge_{ij}$. We ensure progress by building our

solution on the following bound function:

$$\sum_i |p_i - T| \cdot w_i$$

We have now solved the problem of the 'equalities' in our previous example. After assigning weight i to pool $\mathbf{P}_i$ we now have, for example, that communicating a task from $\mathbf{P}_2$ to $\mathbf{P}_1$ decreases the bound function, whereas a communication from $\mathbf{P}_1$ to $\mathbf{P}_0$ increases the bound function. The global effect is that 'equalities' will tend to accumulate from the leaves on up. Our solution, written once again as a global sequential program is:

---

do $(\![i,j : p\text{-}edge_{ij} : p_i > T \ \wedge \ p_j \leq T \rightarrow p_i, p_j := p_i - 1, p_j + 1)$
$[\!]$ $(\![i,j : p\text{-}edge_{ij} : p_i < T \ \wedge \ p_j \geq T \rightarrow p_i, p_j := p_i + 1, p_j - 1)$
od

-Figure 2-

Upon termination, $p_i > T$ in any node, implies $p_{root} > T$, as can easily be proven by induction on the length of a path along *p-edges* from i to the root. Similarly, $p_i < T$ in any node implies $p_{root} < T$. Upon termination we have, therefore $\neg(\exists i,j :: p_i < T \ \wedge \ p_j > T)$ which is equivalent to $(\forall i :: p_i \leq T) \ \vee \ (\forall i :: p_i \geq T)$.

We can choose to assign weights in such a way that it is not guaranteed that there will be no edges for which $w_i = w_j \ \wedge \ edge_{ij}$. If we want to make use of those edges as well, we can use the algorithm in Figure 3. A similar modification can be made to all the versions of the algorithm that we present. However, since it is easy to avoid the existence of such edges by assigning unique weights to the nodes, we have not carried out these modifications.

---

do $(\![i,j : p\text{-}edge_{ij} : p_i > T \ \wedge \ p_j \leq T \rightarrow p_i, p_j := p_i - 1, p_j + 1)$
$[\!]$ $(\![i,j : p\text{-}edge_{ij} : p_i < T \ \wedge \ p_j \geq T \rightarrow p_i, p_j := p_i + 1, p_j - 1)$
$[\!]$ $(\![i,j : edge_{ij} : p_i > T \ \wedge \ p_j < T \rightarrow p_i, p_j := p_i - 1, p_j + 1)$
$[\!]$ $(\![i,j : edge_{ij} : p_i < T \ \wedge \ p_j > T \rightarrow p_i, p_j := p_i + 1, p_j - 1)$
od

-Figure 3-

4. Program transformation

We now transform the global sequential solution into a local and distributed one. We limit ourselves to two processes for now. Process $\mathbf{P}_0$ is the designated root. The only edge in the RDAG is from $\mathbf{P}_1$ to

$\mathbf{P}_0$. In this case our sequential solution simplifies to Figure 4. It describes the combined effect of $\mathbf{P}_0$ and $\mathbf{P}_1$.

$$\begin{array}{l}\textbf{do } p_1 > T \ \wedge \ p_0 \leq T \rightarrow p_1, p_0 := p_1 - 1, p_0 + 1 \\ [\!] \ p_1 < T \ \wedge \ p_0 \geq T \rightarrow p_1, p_0 := p_1 + 1, p_0 - 1 \\ \textbf{od}\end{array}$$

–Figure 4–

Next, we split it into two processes. We introduce new variables q_i to hold the value of the p_i of the other process. Using channels $(OUT_0, IN_1), (OUT_1, IN_0)$ to communicate tasks, we obtain the program in Figure 5 in which we have eliminated all shared variables.

$$\begin{array}{l}\mathbf{P}_0 \equiv \ \{\textit{initialization establishes } q_1 = p_1\} \\ \quad \textbf{do } q_1 > T \ \wedge \ p_0 \leq T \rightarrow IN_1?task; q_1, p_0 := q_1 - 1, p_0 + 1 \\ \quad [\!] \ q_1 < T \ \wedge \ p_0 \geq T \rightarrow OUT_1!task; q_1, p_0 := q_1 + 1, p_0 - 1 \\ \quad \textbf{od}\end{array}$$

$$\begin{array}{l}\mathbf{P}_1 \equiv \ \{\textit{initialization establishes } q_0 = p_0\} \\ \quad \textbf{do } p_1 > T \ \wedge \ q_0 \leq T \rightarrow OUT_0!task; p_1, q_0 := p_1 - 1, q_0 + 1 \\ \quad [\!] \ p_1 < T \ \wedge \ q_0 \geq T \rightarrow IN_0?task; p_1, q_0 := p_1 + 1, q_0 - 1 \\ \quad \textbf{od}\end{array}$$

–Figure 5–

The invariant $p_0 = q_0 \ \wedge \ p_1 = q_1$ guarantees that communications are started under the same condition in both processes. Statement $OUT_1!task$ selects an arbitrary task from the part of the pool stored in $\mathbf{P}_0$ and transmits it via channel OUT_1. Since guard $p_0 \geq T$ holds, the local task pool is nonempty. Statement $IN_1?task$ receives a task via channel IN_1 and adds it to the local part of the task pool. We leave the rest of the verification of this program to the reader.

The next transformation includes the processes $\mathbf{C}_i$. We simulate their effect by including the line

$$\overline{P_i} \rightarrow P_i?p_i$$

where the channel $(., P_i)$ connects each pool process to its computing process. The invariant $p_0 = q_0 \ \wedge \ p_1 = q_1$ of the previous program can now no longer be maintained without extra communication. To

keep this extra communication to a minimum, we keep track of the last communicated values of the p_i in the variables op_i and start a new communication only if a change in p_i affects one of the guards in its neighboring pool process. We cannot predict when another communication may be started by the other process, therefore the communication channels have to be probed. After a communication has started, we must first update the values of the q_i to ensure that communicating a task will indeed result in progress, and to ensure that communications match. This suggests the program shown in Figure 6. To simplify the initialization we start with an empty task pool.

$\mathbf{P}_0 \equiv p_0, q_1, op_0 := 0, 0, 0;$
do $true \rightarrow$
if $\overline{IN_1} \vee$
$(q_1 > T \ \wedge \ p_0 \leq T) \vee (q_1 < T \ \wedge \ p_0 \geq T) \vee$
$(sign(p_0 - T) \neq sign(op_0 - T)) \rightarrow$
$(OUT_1!p_0 \parallel IN_1?q_1); \ op_0 := p_0;$
do $q_1 > T \ \wedge \ p_0 \leq T \rightarrow IN_1?task; q_1, p_0, op_0 := q_1 - 1, p_0 + 1, op_0 + 1$
$[\!] \ q_1 < T \ \wedge \ p_0 \geq T \rightarrow OUT_1!task; q_1, p_0, op_0 := q_1 + 1, p_0 - 1, op_0 - 1$
od
$[\!] \ \overline{P_0} \ \rightarrow P_0?p_0$
fi
od

$\mathbf{P}_1 \equiv q_0, p_1, op_1 := 0, 0, 0;$
do $true \rightarrow$
if $\overline{IN_0} \vee$
$(p_1 > T \ \wedge \ q_0 \leq T) \vee (p_1 < T \ \wedge \ q_0 \geq T) \vee$
$(sign(p_1 - T) \neq sign(op_1 - T)) \rightarrow$
$(OUT_0!p_1 \parallel IN_0?q_0); \ op_1 := p_1;$
do $p_1 > T \ \wedge \ q_0 \leq T \rightarrow OUT_0!task; p_1, q_0, op_1 := p_1 - 1, q_0 + 1, op_1 - 1$
$[\!] \ p_1 < T \ \wedge \ q_0 \geq T \rightarrow IN_0?task; p_1, q_0, op_1 := p_1 + 1, q_0 - 1, op_1 + 1$
od
$[\!] \ \overline{P_1} \ \rightarrow P_1?p_1$
fi
od

-Figure 6-

We prove the generalization of this algorithm to the RDAG described in section 3. The program can be found in Figure 7. The channels are (OUT_{ij}, IN_{ji}), for which $p\text{-}edge_{ij} \vee p\text{-}edge_{ji}$. For those i and j we have introduced variables op_{ij} to keep track of the value of p_i last communicated from process $\mathbf{P}_i$ to process $\mathbf{P}_j$ and variables q_{ij} to hold the values of the p_j last communicated by process $\mathbf{P}_j$.

$\mathbf{P}_i \equiv$

$p_i := 0;\ (;\ j : p\text{-}edge_{ij} \vee p\text{-}edge_{ji} : q_{ij}, op_{ij} := 0, 0);$

do $true \rightarrow$

if $(\llbracket j : p\text{-}edge_{ji} :$

$\overline{IN_{ij}} \vee$

$(q_{ij} > T \ \wedge \ p_i \leq T) \vee (q_{ij} < T \ \wedge \ p_i \geq T) \vee$

$(sign(p_i - T) \neq sign(op_{ij} - T)) \rightarrow$

$(OUT_{ij}!p_i \parallel IN_{ij}?q_{ij});\ op_{ij} := p_i;$

do $q_{ij} > T \ \wedge \ p_i \leq T \rightarrow IN_{ij}?task; q_{ij}, p_i, op_{ij} := q_{ij} - 1, p_i + 1, op_{ij} + 1$

$\llbracket \ q_{ij} < T \ \wedge \ p_i \geq T \rightarrow OUT_{ij}!task; q_{ij}, p_i, op_{ij} := q_{ij} + 1, p_i - 1, op_{ij} - 1$

od

)

$\llbracket \ (\llbracket j : p\text{-}edge_{ij} :$

$\overline{IN_{ij}} \vee$

$(p_i > T \ \wedge \ q_{ij} \leq T) \vee (p_i < T \ \wedge \ q_{ij} \geq T) \vee$

$(sign(p_i - T) \neq sign(op_{ij} - T)) \rightarrow$

$(OUT_{ij}!p_i \parallel IN_{ij}?q_{ij});\ op_{ij} := p_i;$

do $p_i > T \ \wedge \ q_{ij} \leq T \rightarrow OUT_{ij}!task; p_i, q_{ij}, op_{ij} := p_i - 1, q_{ij} + 1, op_{ij} - 1$

$\llbracket \ p_i < T \ \wedge \ q_{ij} \geq T \rightarrow IN_{ij}?task; p_i, q_{ij}, op_{ij} := p_i + 1, q_{ij} - 1, op_{ij} + 1$

od

)

$\llbracket \ \overline{P_i} \ \rightarrow P_i?p_i$

fi

od

-Figure 7-

We need the following invariant: $(\forall i, j :: q_{ij} = op_{ji})$. The invariant holds initially and continues to hold because the statement $(OUT_{ij}!p_i \parallel IN_{ij}?q_{ij});\ op_{ij} := p_i$ in process $\mathbf{P}_i$ pairs with statement $(OUT_{ji}!p_j \parallel IN_{ji}?q_{ji});\ op_{ji} := p_j$ in process $\mathbf{P}_j$ to yield the assignment $op_{ij}, q_{ji}, op_{ji}, q_{ij} := p_i, p_i, p_j, p_j$ which clearly maintains the invariant. Furthermore assignment $q_{ij}, p_i, op_{ij} := q_{ij} - 1, p_i + 1, op_{ij} + 1$ in the

first alternative of the first innermost **do** construct of process $\mathbf{P}_i$ is guaranteed to match with assignment $p_i, q_{ij}, op_{ij} := p_i - 1, q_{ij} + 1, op_{ij} - 1$ in the second innermost **do** construct in process $\mathbf{P}_j$ because the preceding statements establish $p_i = q_{ji} \ \wedge \ p_j = q_{ij}$. The other alternative is similar and both pairs maintain the invariant. Finally, statement $P_i?p_i$ does not change q_{ij} or op_{ij}.

We check our three requirements one by one:

(0)'. Either there are communications pending or $(\forall i :: p_i \leq T) \ \vee \ (\forall i :: p_i \geq T)$.

We assume all processes have been suspended on the guards in the **if** statements and that $(\exists i, j :: p_i < T \ \wedge \ p_j > T)$ and derive a contradiction as in section 3. If $p_i < T$ and all guards are false, then $(\forall j : p\text{-}edge_{ij} : q_{ij} < T)$. Unless i is the root, such a j exists. The invariant implies $op_{ji} < T$ for that j and the falsity of the guard $sign(p_j - T) \neq sign(op_{ji} - T)$ implies $p_j < T$. By induction $p_{root} < T$. By symmetry we also have $p_{root} > T$, which establishes the contradiction.

(0)". There is no deadlock.

The graph we have constructed is a (palm-)tree. Since this is an acyclic structure, and since the algorithm guarantees that two processes cannot deadlock when they are committed to communicating with one another, there is no cycle in the communication dependencies. Therefore there is no deadlock.

(1). If the processes $\mathbf{C}_i$ do not consume or produce tasks, the number of communications is bounded.

A bound function that decreases on every repetition of the outermost **do** construct (other than communications between a $\mathbf{P}_i$ and a $\mathbf{C}_i$) is

$$\sum_i |p_i - T| \cdot w_i + (\mathbf{N}i, j : p{-}edge_{ij} \vee p{-}edge_{ji} : p_i \neq q_{ji})$$

The proof is as follows. An execution of the first alternative in process $\mathbf{P}_i$ always matches with the execution of the second alternative in some process $\mathbf{P}_j$ for which $p\text{-}edge_{ji}$. For any such pair the first sum can never increase due to the fact that the first pair of communications establishes $p_i = q_{ji} \ \wedge \ p_j = q_{ij}$. If before the communication either $(sign(p_i - T) \neq sign(op_{ij} - T))$ in $\mathbf{P}_i$ or $(sign(p_j - T) \neq sign(op_{ji} - T))$ in $\mathbf{P}_j$, then the assignments $op_{ij} := p_i$ and $op_{ji} := p_j$ in $\mathbf{P}_i$ and $\mathbf{P}_j$ respectively, in conjunction with the invariant $op_{ij} = q_{ji} \ \wedge \ op_{ji} = q_{ij}$, imply that the variant function decreases. If $(sign(p_i - T) = sign(op_{ij} - T))$ in $\mathbf{P}_i$ and $(sign(p_j - T) = sign(op_{ji} - T))$ in $\mathbf{P}_j$, but all guards in the innermost **do** statements evaluate to false, so that the first part of the variant function does not decrease, either q_{ij} in $\mathbf{P}_i$ or q_{ji} in $\mathbf{P}_j$ must have changed in the first pair of communications and the second term in the invariant has decreased.

Note. The algorithm in Figure 7 can be somewhat simplified by omitting $(p_i > T \ \wedge \ q_{ij} \leq T) \vee (p_i > T \ \wedge \ q_{ij} \geq T)$ from the guard of the second outermost alternative. This can be done because for any pair $\mathbf{P}_i, \mathbf{P}_j$ of processes that may communicate the following holds:

$$\begin{aligned}
&(p_i > T \ \wedge \ q_{ij} \leq T) \vee (p_i < T \ \wedge \ q_{ij} \geq T) \\
&\Rightarrow \\
&(q_{ji} > T \ \wedge \ p_j \leq T) \vee (q_{ji} < T \ \wedge \ p_j \geq T) \vee \\
&sign(p_i - T) \neq sign(op_{ij} - T) \vee \\
&sign(p_j - T) \neq sign(op_{ji} - T)
\end{aligned}$$

which can be proven by using the invariant $op_{ij} = q_{ji} \ \wedge \ op_{ji} = q_{ij}$.

5. More thresholds

If we want to put stronger bounds on how much the pools can be allowed to differ, we can add more thresholds in a straightforward manner. Condition $(\forall i :: p_i \leq T) \vee (\forall i :: p_i \geq T)$ is replaced by $(\forall k :: (\forall i :: p_i \leq T_k) \vee (\forall i :: p_i \geq T_k))$. The algorithm is given in Figure 8.

$\mathbf{P}_i \equiv$

$p_i := 0;\ (;\ j : p\text{-}edge_{ij} \vee p\text{-}edge_{ji} : q_{ij}, op_{ij} := 0, 0);$

do $true \rightarrow$

if $(\llbracket j : p\text{-}edge_{ji} :$

$\overline{INij} \vee$

$(\exists k :: (q_{ij} > T_k \ \wedge \ p_i \leq T_k) \vee (q_{ij} < T_k \ \wedge \ p_i \geq T_k)) \vee$

$(\exists k :: sign(p_i - T_k) \neq sign(op_{ij} - T_k)) \rightarrow$

$(OUT_{ij}!p_i \parallel IN_{ij}?q_{ij});\ op_{ij} := p_i;$

do $(\exists k :: q_{ij} > T_k \ \wedge \ p_i \leq T_k) \rightarrow IN_{ij}?task; q_{ij}, p_i, op_{ij} := q_{ij} - 1, p_i + 1, opij + 1$

$\llbracket \ (\exists k :: q_{ij} < T_k \ \wedge \ p_i \geq T_k) \rightarrow OUT_{ij}!task; q_{ij}, p_i, op_{ij} := q_{ij} + 1, p_i - 1, op_{ij} - 1$

od

$)$

$\llbracket \ (\llbracket j : p\text{-}edge_{ij} :$

$\overline{IN_{ij}} \vee$

$(\exists k :: (p_i > T_k \ \wedge \ q_{ij} \leq T_k) \vee (p_i < T_k \ \wedge \ q_{ij} \geq T_k)) \vee$

$(\exists k :: sign(p_i - T_k) \neq sign(op_{ij} - T_k)) \rightarrow$

$(OUT_{ij}!p_i \parallel IN_{ij}?q_{ij});\ op_{ij} := p_i;$

do $(\exists k :: p_i > T_k \ \wedge \ q_{ij} \leq T_k) \rightarrow OUT_{ij}!task; p_i, q_{ij}, op_{ij} := p_i - 1, q_{ij} + 1, op_{ij} - 1$

$\llbracket \ (\exists k :: p_i < T_k \ \wedge \ q_{ij} \geq T_k) \rightarrow IN_{ij}?task; p_i, q_{ij}, op_{ij} := p_i + 1, q_{ij} - 1, op_{ij} + 1$

od

$)$

$\llbracket \ \overline{P_i} \ \rightarrow P_i?p_i$

fi

od

–Figure 8–

The proof that our first two requirements are still being met is nearly identical to the case with one threshold and is left to the reader. However, to guarantee that the number of communications, on the

absence of communications with the computing processes, is bounded, we have to insist that the threshold values differ by at least two. A bound function is then given by the following two-tuple, and the ordering is lexicographic ordering.

$$(\sum_i p_i^2 + (Ni,j : p{-}edge_{ij} \vee p{-}edge_{ji} : p_i \neq q_{ji}), \sum_i (MINk :: |p_i - T_k| \cdot w_i))$$

Without proof we state that, as in the previous algorithm, on every communication between two processes $\mathbf{P}_i$ and $\mathbf{P}_j$, either a task is being communicated, or the second term in the first element of the two-tuple decreases. If a task is being communicated, and p_i and p_j differed originally by more than one, the sum over the squares decreases. Without loss of generality we can state $w_i < w_j$, therefore $p\text{-}edge_{ji} = true$. If p_i and p_j originally differed by exactly one, then it follows that originally $p_i = T_k$ for some k and $|p_j - T_k| = 1$ for that k. The communication of the task then reduces the second element of the two-tuple by $w_j - w_i$. The fact that the T_k differ by at least two, is needed here to guarantee that the k's for which the terms in the second element of the two tuple are minimal do not change.

We can greatly reduce the amount of computation needed to evaluate the guards in the algorithm in Figure 8 by maintaining variables tl_i and tg_i such that

$$tl_i \leq p_i \leq tg_i \ \wedge \ ((\exists k :: tl_i = T_k) \vee tl_i = 0) \ \wedge \ ((\exists k :: tg_i = T_k) \vee tg_i = poolsize) \ \wedge$$
$$\neg(\exists k :: tl_i < T_k \leq p_i \vee p_i \leq T_k < tg_i)$$

All existential quantifications can then be removed from the guards.

6. Finite pool sizes

If the size of the pools is fixed, it is possible that our algorithm attempts to add a task to a pool that has reached its capacity. Not much can be done about the situation where all pools are full, but it would be problematic if the program would deadlock in a situation where pool space is available in another processor. Since a communication between two pool processes does not increase the size of the largest pool, a problem can only occur in the communications between the $\mathbf{C}_i$ and their $\mathbf{P}_i$. If we replace the single channel by two channels, one over which tasks are consumed, and one over which tasks are produced, and replace the guard for P_{prod} by $\overline{P}_{prod} \ \wedge \ p_i < poolsize$ and assume that tasks are produced one at a time, the problem is solved. We can ensure that computation power is used efficiently until all buffers fill up by choosing one of the thresholds to be $poolsize - 1$.

7. Discussion

This paper is an attempt to deal in a systematic way with a problem that in our experience occurs frequently. In the past when solving a distributed programming problem of this type, we would construct the $\mathbf{C}_i$ for that particular problem, think of a strategy that would provide a decent load balance, and construct $\mathbf{P}_i$ for that specific problem. The present solution seems to be more general.

Acknowledgement

We thank Nan Boden for helpful comments on the manuscript.

References

[1] E.W. Dijkstra, *A Discipline of Programming*, (Prentice Hall, Englewood Cliffs, NJ 1976).

[2] C.A.R. Hoare, Communicating Sequential Processes, *Comm. ACM* (1978) 666-677.

[3] C.A.R. Hoare, *Communicating Sequential Processes*, (Prentice-Hall International Series in Computer Science, 1985)

[4] A.J. Martin, An Axiomatic Definition of Synchronization Primitives, *Acta Informatica* 16, (1981) 219-235.

[5] S. Owicki, D. Gries, An Axiomatic Proof Technique for Parallel programs, *Acta Informatica* 6, (1976) 319-340.

[6] H. P. Hofstee, A. J. Martin, J. L. A. van de Snepscheut, Distributed Sorting, *Science of Computer Programming* 15 (1990) 119-133

[7] A.J. Martin, The Probe: An Addition to Communication Primitives, *Information Processing Letters* 20 (1985) 125-130